CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

A Statistical Hand Gesture Recognition System Using the Leap Motion Controller

A graduate thesis submitted in partial fulfillment of the requirements

For the degree of Master of Science in Computer Science

By

Michael DiMartino

May 2016

The graduate project of Michael DiMartino is approved:

_____	_____
Dr. Jeff Wiegley	Date
_____	_____
Dr. Robert McIlhenny	Date
_____	_____
Dr. G. Michael Barnes, Chair	Date

California State University, Northridge

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

A Statistical Hand Gesture Recognition System Using the Leap Motion Controller

By

Michael DiMartino

Master of Science in Computer Science

As technology continues to improve, hand gesture recognition as a form of human-computer interaction is becoming more and more feasible. One such piece of technology, the Leap Motion Controller, provides 3D tracking data of the hands through an easy-to-use API. This thesis presents an application that uses Leap Motion tracking data to learn and recognize static and dynamic hand gestures. Gestures are recognized using statistical pattern recognition. Each gesture is defined by a set of features including fingertip positions, hand orientation, and movement. Given sufficient training data, the similarity between two gestures is measured by comparing each of their corresponding features. Two separate implementations are presented for dealing with the temporal aspect of dynamic gestures. Users are able to interact with the system using a convenient graphical user interface. The accuracy of the system was experimentally tested with the help of two separate test participants: one for the training phase and one for the recognition phase. All test gestures (both static and dynamic) were successfully recognized with minimal training data. In some cases, additional gestures were mistakenly recognized.

INTRODUCTION

Overview

Body language has always been one of the most basic, universal forms of communication between human beings in everyday life. Hand gestures in particular can express a very wide range of different ideas. Considering that a person has two hands, each hand with five fingers, and each finger with multiple joints; there is a high number of different static hand gestures that can be identified by the various configurations. An even higher number of dynamic gestures can be distinguished when we consider sequences of static gestures and factor in movement and speed. Entirely functional languages with complex grammar systems (sign languages) have been defined using gestures [1,2].

As technology has improved over the years, it is becoming more and more possible for humans to communicate with computers through hand gestures. While devices like the mouse and keyboard are practical to use, they reduce the naturalness of human-computer interaction (HCI). This limitation has become more evident with the emergence of virtual reality [3]. It is more intuitive to interact with objects and other people in a virtual world using one's hands. Even in the real world, using gesture input to control consumer devices is quite practical. TVs, sound systems, and other household appliances could be controlled by gesture input. One could be sitting on the couch watching TV, and flip through channels by waving their hand left or right, adjust the volume by raising their hand up or down, and so forth; all without ever picking up a remote.

This paper discusses a system for recognizing static and dynamic hand gestures using input from the Leap Motion Controller [4]. The user is able to define and edit custom gestures to be recognized by the application.

Background and Related Work

Much of the existing work in gesture recognition has been limited by the technology available at the time. Early work started with the invention of glove based control interfaces. These early "data gloves" were essentially wired interfaces with sensors attached along the fingers, able to detect the bending of different joints. With no pre-processing required, the simple design of such gloves was ideal for the limited processing power available at the time. Though accurate, the gloves tend to be expensive and cumbersome [6]. While wearing a glove, the user cannot easily type on a keyboard or manipulate any other tools or objects [5]. Data gloves have improved since then: newer designs are wireless and more convenient to wear. Two distinct categories of data gloves have emerged over the years: (1) *Active data gloves* are made up of multiple sensors that measure acceleration or flexing of joints and communicate the data to the hosting device via wired or wireless technology. (2) *Passive data gloves* are much simpler: markers or colors are placed on each of the fingers for detection by a camera, with no sensors used [7]. Some of the earliest data glove prototypes include the Sayre Glove, the MIT-LED Glove, and the Digital Entry Data Glove. Commercially available products were eventually created, including Power Glove, P5 Glove, CyberGlove, and Super Glove [8].

Vision based approaches started not much later, though they were limited by technical obstacles of the time. Along with the low computing power available, cameras offered

poor resolution and color inconsistency. Despite these issues, a working vision based gesture recognition system was reported in the early 1980s [7]. The use of LEDs or bright colors to identify the fingertips and different regions of the hands was often necessary in these approaches. Ongoing improvements in computer vision have produced technologies like the Leap Motion Controller and the Kinect [9], which are able to produce accurate 3D tracking data without the need for any extra equipment. Both devices process image data to create skeletal models. Leap Motion focuses on just the hands (see Figure 1), while Kinect tracks the entire body (see Figure 2).

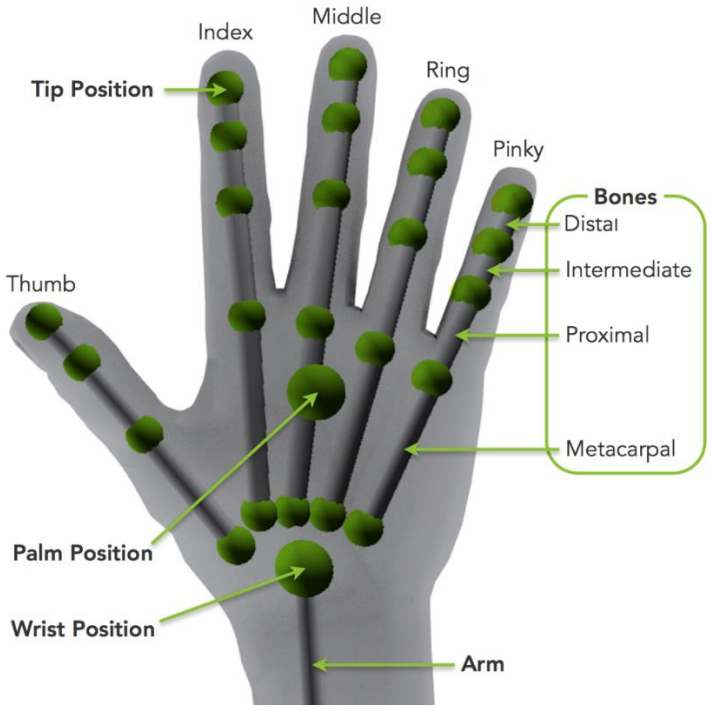

Figure 1 - Leap Motion hand skeleton [10]

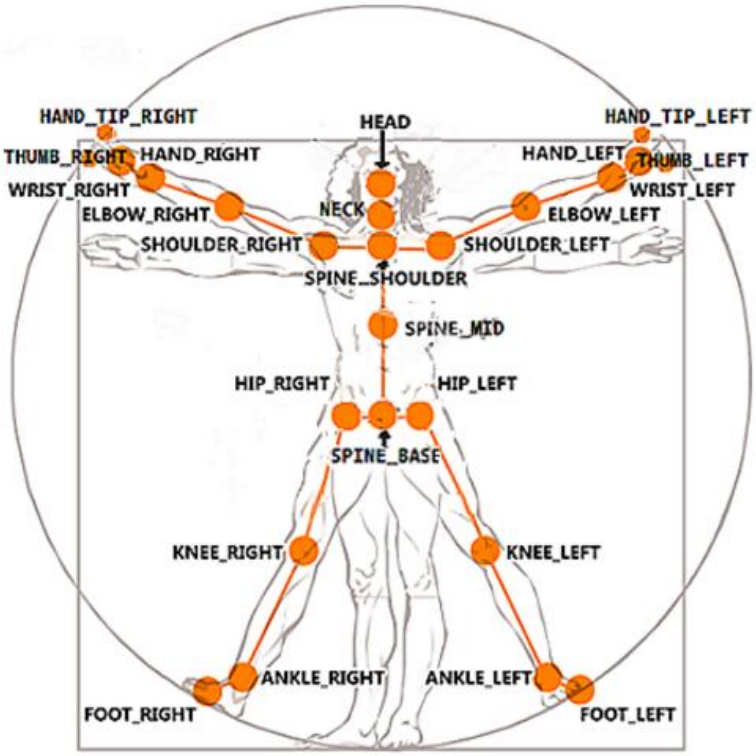

Figure 2 - Kinect skeleton data joints [9]

Much work has already been done in the field of hand gesture recognition. No shortage of papers is available online discussing the details of such systems. In [22], Japanese kana alphabet gestures are recognized with a data-glove based system using joint angle and hand orientation coding techniques. In [24], Newby describes a statistical based gesture recognition system that uses a “sum of squares” template matching approach to recognize American Sign Language (ASL) signs. The system employs a DataGlove to measure finger position. In [25], Hong describes an approach to 2D gesture recognition that models each gesture as a finite state machine (FSM) in spatial-temporal space [23]. In [20], the Leap Motion and Kinect devices are used in tandem to produce a more precise feature set. The relevant features from both devices are combined and fed into a multi-

class Support Vector Machine (SVM) classifier in order to recognize the performed gestures. In [21], the strengths and weaknesses of the Leap Motion Controller are analyzed, and its suitability for recognizing Australian Sign Language (Auslan) is explored. They found that there is potential for using the technology to recognize Auslan, but further development of the Leap Motion API is required (this paper was written in 2013; the Leap Motion API has improved since then).

Objective

The objective of this project is to create a system for recognizing static and dynamic hand gestures using statistical pattern recognition. The effectiveness of this system is demonstrated through a desktop application that uses input from the Leap Motion Controller to recognize and learn new hand gestures. The user is able to browse existing gestures and perform the following actions on them: view, edit, or delete. When not viewing a gesture, a live 3D rendering of the hands is displayed on screen.

PROJECT DETAILS

Overview

The application was built using WPF (Windows Presentation Framework), with C# as the programming language and XAML (Extensible Application Markup Language) as the user interface design language [33]. The application uses SharpGL for 3D graphics [12], SQLite for saving gestures and other configuration options [14], and the Leap Motion API for hand tracking data [18]. Figure 3 shows how all of these components work together.

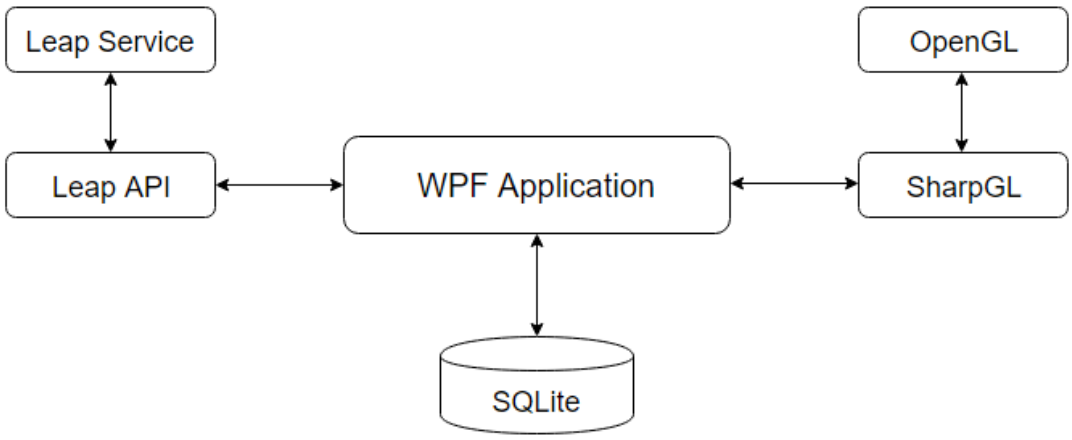

Figure 3 - High-level API architecture diagram

In the context of this project, hand gestures are divided into two categories: static and dynamic. A static gesture is a snapshot of one or both of the hands. No movement is involved. Any single configuration of the hands is a static gesture. Dynamic gestures can be thought of as a sequence of static gestures, with movement throughout. Examples include clapping, waving, or throwing a punch. The system supports both one- and two-handed gestures.

Tools and Technologies

WPF (Windows Presentation Foundation)

WPF is a GUI framework developed by Microsoft using the .NET framework. WPF allows users to develop user interfaces with both markup (XAML) and code-behind (C# or VB). XAML is used to implement the application's appearance declaratively (windows, buttons, textboxes, etc.). The code-behind implements the functionality for responding to user interactions, including event handling and calling the appropriate business / data access logic in response [11].

The project was designed with the Model-View-ViewModel (MVVM) pattern in mind, a well suited design pattern for WPF applications. The Model is the domain object, including data models along with business and validation logic. The View is responsible for presentation of the data, and initiating commands in response to user input. It should be defined almost entirely in XAML, with minimal code-behind. The ViewModel acts as a mediator between the Model and the View. It retrieves data from the Model and makes it available to the View, possibly reformatting the data in a way that is easier for the View to handle. It is also responsible for the implementation of commands that were initiated in the View. Proper use of the MVVM pattern keeps the application logic and user interface separate, making the project easier to manage and maintain in the long run [34]. All Model and ViewModel code was written in C# (along with the View code-behind).

SharpGL

SharpGL is a C# library that allows for easy integration of OpenGL into a .NET Framework based application [12]. It wraps OpenGL (Open Graphics Library) and GLU (OpenGL Utility Library). The hands, references axes, and all other 3D graphics are drawn using SharpGL. All entities are built with cylinders and spheres, using methods that wrap GLU functions including `gluNewQuadric()`, `gluCylinder()` and `gluSphere()`. SharpGL allows an OpenGL window to be used as a WPF control. The control triggers the `OpenGLDraw` event at regular intervals based on its configured `FrameRate`. This draw event is essentially the "heartbeat" or "clock signal" of the application. A new frame is requested from the Leap Motion Controller on each draw event, and then processed by the application accordingly.

SQLite

SQLite is an embedded SQL database engine that reads and writes directly to ordinary disk files. It provides local data storage for the application, including gesture and configuration data. The application interacts with the database via the `System.Data.SQLite` library, an ADO.NET provider for SQLite [26]. There was no real need for a relational database in this project, but SQLite was chosen out of convenience. Table 1 describes each of the tables that make up the database. All gestures are serialized to JSON before being written to the database.

Table 1 – SQLite database tables

Table Name	Columns	Description
BoneColors	bone (TEXT, PRIMARY KEY) color (TEXT)	Stores the color of each bone that the user has set in the "Bone Colors" options menu.
BoolOptions	name (TEXT, PRIMARY KEY) value (INTEGER)	Stores boolean options that the user has set in "General Options".
StaticGestureClasses	id (INTEGER, PRIMARY KEY) name (TEXT) gesture_json (TEXT) sample_instance_json (TEXT)	Stores static gesture classes. The 'gesture_json' column stores the actual StaticGesture object. The 'sample_instance_json' column stores the default gesture instance that should be displayed when the gesture is viewed.
StaticGestureInstances	id (INTEGER, PRIMARY KEY) class_id (INTEGER, FOREIGN KEY) json (TEXT)	Stores individual instances of static gestures. The 'class_id' column references the class it belongs to in the StaticGestureClasses table. The 'json' column stores the actual StaticGestureInstance object.
DynamicGestureClasses	id (INTEGER, PRIMARY KEY) name (TEXT) gesture_json (TEXT) sample_instance_json (TEXT)	Same as StaticGestureClasses, but for dynamic gestures.
DynamicGestureInstances	id (INTEGER, PRIMARY KEY) class_id (INTEGER, FOREIGN KEY) json (TEXT)	Same as StaticGestureInstances, but for dynamic gestures.

Leap Motion Controller (and its API)

The Leap Motion Controller tracks the hands and generates 3D tracking data. The small device can be placed on top of a desk, facing upward, or mounted to a VR headset facing forward. It tracks hands at up to 200 frames per second and gives a 150° field of view with approximately 8 cubic feet of interactive 3D space [15].

The device consists of two cameras and three infrared LEDs. The cameras track infrared light with a wavelength of 850 nanometers, which is outside the visible light spectrum. The data takes the form of a stereo image (one for each camera) in which the only visible objects are those that are illuminated by infrared light. Since the Leap Motion Controller's viewing range is limited to approximately 2 feet (60 cm) above the device, only the hands should be visible [16]. Figure 4 depicts the Leap Motion Controller's interaction space with the controller placed face up, in front of the keyboard.

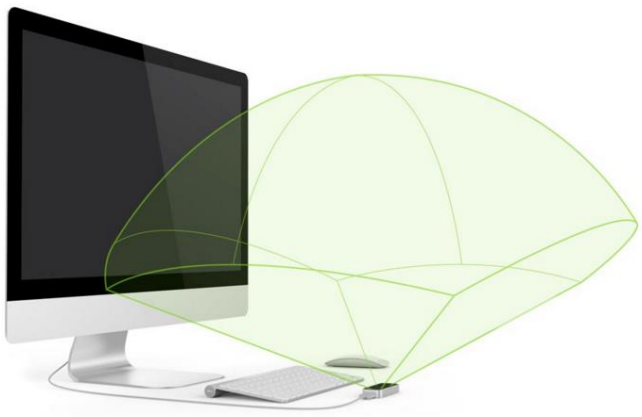

Figure 4 - The Leap Motion Controller and its interaction area [15]

The Leap Motion Service runs on the user's computer and processes the infrared images it receives from the device. After compensating for irrelevant background objects and other noise, the two images are analyzed to reconstruct a 3D representation of what the device has seen. Next, the software extracts tracking data from this 3D model using proprietary algorithms that are able to infer the position of regions that may be occluded by other objects. For example, if the view of one finger is obstructed by the palm and/or other fingers, the Leap Motion software is able to make an educated (but not always correct) guess as to how that finger is positioned. Finally, this tracking data is organized

into an object-oriented API in the form of `Frame` objects, which contain information about the hands in one snapshot of time [16]. "The `Frame` object is essentially the root of the Leap Motion data model" [36]. Applications can access this data using the Leap Motion API, either by manually requesting the most recent `Frame` or by setting up a listener to receive `Frame` objects as they become available. In this project, `Frame`s are requested at the configured SharpGL frame rate of 30 frames / second. Table 2 provides information about the Leap Motion API classes used in this project.

Table 2 - Leap Motion API: Classes used in this project (SDK version 2.3). All information taken from [18].

Class	Properties	Methods	Description
`Arm`	8	3	Represents the forearm.
`Bone`	10	3	Represents a tracked bone.
`Config`	0	11	Provides access to Leap Motion system configuration information.
`Controller`	6	18	The main interface to the Leap Motion Controller.
`Finger`	2	5	Represents a tracked finger.
`FingerList`	5	4	Represents a list of `Finger` objects.
`Frame`	13	20	Represents a set of hand and finger tracking data detected in a single frame.
`Hand`	25	14	Reports the physical characteristics of a detected hand.
`HandList`	5	2	Represents a list of `Hand` objects.
`Listener`	0	15	Defines a set of callback functions that can be overridden in a subclass to respond to events dispatched by the `Controller` object.
`Matrix`	5	19	Represents a transformation matrix.
`Pointable`	17	3	Reports the physical characteristics of a detected finger or tool.

`PointableList`	5	5	Represents a list of `Pointable` objects.
`Vector`	19	17	Represents a direction or position in three-dimensional space.

The Leap Motion API measures distance in millimeters, time in microseconds (unless otherwise noted), speed in millimeters / second, and angles in radians [36].

Static Gesture Recognition

Both static and dynamic gesture recognition were implemented using a statistical pattern recognition approach. A pattern is represented by a set of *d* features, or attributes, viewed as a *d*-dimensional feature vector [17]. In this case, the feature vector consists of a subset of the data available in one `Frame`. Table 3 shows the features used in static gesture recognition.

Table 3 - Features used in static gesture recognition

Name	Details
Pitch	*Type*: `float` *Description*: The hand's rotation about the x-axis in radians.
Yaw	*Type*: `float` *Description*: The hand's rotation about the y-axis in radians.
Roll	*Type*: `float` *Description*: The hand's rotation about the z-axis in radians.
Pinky Tip Position	*Type*: `Vec3` *Description*: Position of the pinky fingertip relative to the hand's object space.
Ring Tip Position	*Type*: `Vec3` *Description*: Position of the ring fingertip relative to the hand's object space.
Middle Tip Position	*Type*: `Vec3` *Description*: Position of the middle fingertip relative to the hand's object space.
Index Tip Position	*Type*: `Vec3` *Description*: Position of the index fingertip relative to the hand's object space.

Thumb Tip Position	*Type*: `Vec3` *Description*: Position of the thumb tip relative to the hand's object space.

The features in Table 3 are for a single hand. Thus, a one-handed gesture is represented by an 8-dimensional feature vector, while a two-handed gesture is represented by a 16-dimensional feature vector. The "Pitch" and "Yaw" features are computed using the hand's direction vector (from palm center to fingers), and the "Roll" feature is computed using the hand's palm normal vector. These three features are of type `float` and their values range between $-\pi$ and π. The fingertip position features are of type `Vec3` and are relative to the hand's object space, with the origin at the center of the palm. The pitch axis points sideways across the hand, the roll axis points forward (palm center to fingers), and the yaw axis is parallel with the palm normal [18]. Each value is scaled based on the size of the user's finger (measured at run-time) so that a fully extended finger has a tip position magnitude of 1, regardless of the actual finger size.

Before any recognition can take place, the system must acquire some training data. This is done using a supervised learning approach, where the user first identifies which gesture they will perform and then performs it one or more times. It is up to the user to decide how many instances of the gesture to record. A sample size of 5-10 is usually sufficient, as long as there are enough instances to accurately represent the acceptable range of variations of that gesture. All instances are saved to the database. These instances are processed to produce a gesture class, containing the mean and standard deviation values

for each feature. The mean value of a `Vec3` feature is of type `Vec3`, while its standard deviation is of type `float` (Euclidean distance from the mean).

With some gesture classes defined, the system can compare new gesture instances to existing classes to try and find a match. This is accomplished by computing the "distance" (in 8- or 16-space, depending on the number of hands) between the instance and each of the classes. Figure 5 shows the pseudocode for how distance is calculated (without feature weighting).

```
float Distance(gestureInstance, gestureClass) {
    // HandConfiguration: NoHands, LeftHandOnly, RightHandOnly, or BothHands
    if(gestureInstance.HandConfiguration != gestureClass.HandConfiguration) {
        return PositiveInfinity;
    }

    float distance = 0;
    int featureCount = gestureInstance.FeatureVector.Count;
    for(int i=0; i < featureCount; i++) {
        float mean = gestureClass.MeanFeatureVector[i];
        float stdDev = gestureClass.StdDevFeatureVector[i];
        distance += gestureInstance.FeatureVector[i].DistanceTo(mean) / stdDev;
    }

    return distance / (float)featureCount;
}
```

Figure 5 - Pseudocode for calculating distance between static gesture instance and class (no feature weighting)

The distance is the average number of standard deviations each feature is from the mean. For this project, a distance of 3.0 or less is considered a match. Why 3.0? For a normal distribution, 99.7% of the data lies within 3 standard deviations of the mean [28]. We assume that each gesture's feature vector is multivariate normally distributed (8- or 16-variate depending on the number of hands). Thus, any gesture instance with distance 3.0 or less from some gesture class is considered a member of that class.

Dynamic Gesture Recognition

A dynamic gesture can be thought of as a sequence of static gestures. One or both of the hands are moving, possibly changing posture throughout. This presents a new problem: How to identify the start and end of a dynamic gesture? For the purposes of this project, it is assumed that the user's hands are still for a short interval of time before and after the gesture is performed. This way guarantees that the start and end samples are accurate.

Recognition of dynamic gestures is more complicated than it is for static gestures, because of the temporal aspect: each dynamic gesture instance is a time series of gesture samples. These samples are essentially static gesture instances, but also include velocity information (hand speed and direction). To compute the distance between dynamic gestures, we must compare sequences of static gesture instances (plus velocity). This presents a challenge, as no two instances are exactly the same, even if they represent the same gesture. One is likely to have a different number of samples than the other (varying time or speed), or include sporadic accelerations and decelerations throughout. To deal with this problem, two separate approaches were implemented:

(1) Lerp: Use linear interpolation to map samples of one sequence directly to the other.

(2) DTW: Use dynamic time warping (DTW) to align the sequences.

Lerp

The linear interpolation approach works best for sequences that, when resized to have the same sample count, are in phase with one another. One is essentially a slowed-down version of the other. Figure 6 shows an example of this.

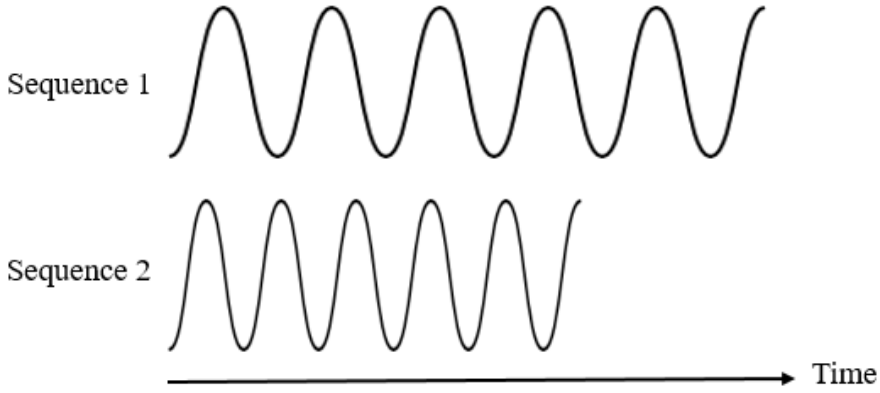

Figure 6 - Slow and fast versions of the same sequence

If we split both sequences in Figure 6 into *N* evenly spaced samples each, we would see that samples at matching indexes are identical. Sequences aren't always going to align perfectly like in Figure 6-- that's a best case scenario. We can think about these sequences as dynamic gesture instances, which are really just sequences of static gestures. Consider the following: We have two instances of the same dynamic gesture. One is made up of 4 samples while the other is made up of 6. We want to compare the two instances, but how do we map samples from the first instance to the other? In other words, how can we condense the 6 sample instance down to 4 samples (or vice versa), so that we have a one-to-one mapping of samples between the two gesture instances? We choose to map the first (last) sample in instance 1 to the first (last) sample in instance 2, and space the rest evenly between. Table 4 shows how samples from the 4-sample instance map to samples in the 6-sample instance (with index starting at 0).

Table 4 - Indexes of 4-sample instance mapped to 6-sample instance

Index in Instance 1	Index in Instance 2
0	0
1	1.67
2	3.34
3	5

So, if we want to measure the distance between these two dynamic gesture instances, we compare `instance1.sample[0]` to `instance2.sample[0]`, then `instance1.sample[1]` to `instance2.sample[1.67]`, and so on. We compute the distance between each of these pairs, sum them up, and divide by the total number of distances (4 in this case) to get the average distance between corresponding samples. But how do we know what `instance2.sample[1.67]` is? We know the values of `instance2.sample[1]` and `instance2.sample[2]`, so we can interpolate between these two samples by 67% to get `instance2.sample[1.67]`. This requires the implementation of a method of the form:

```
DGISample Lerp(DGISample sample1, DGISample sample2, float amount);
```

The method name `Lerp` is short for linearly interpolate, `DGISample` is the type of a dynamic gesture instance sample, and `amount` is the amount to interpolate by (clamped to the range [0, 1]). Using this method, we can resize the 6 samples down to 4 (though some accuracy is lost), and compute the distance between the two instances using the method shown in Figure 7 (which assumes the two dynamic gesture instances have the same number of samples). The same concept can be applied to dynamic gesture classes.

```
float Distance(dg1, dg2) {
  float distance = 0;
  int numSamples = dg1.Samples.Count;
  for(int i=0; i < numSamples; i++) {
    distance += dg1.Samples[i].DistanceTo(dg2.Samples[i]);
  }
  return distance / (float)numSamples;
}
```

Figure 7 - Pseudocode for calculating distance between dynamic gestures of the same sample size

Dynamic Time Warping

Dynamic time warping (DTW) is a well-known technique for finding an optimal alignment between two given (time-dependent) sequences (see Figure 8).

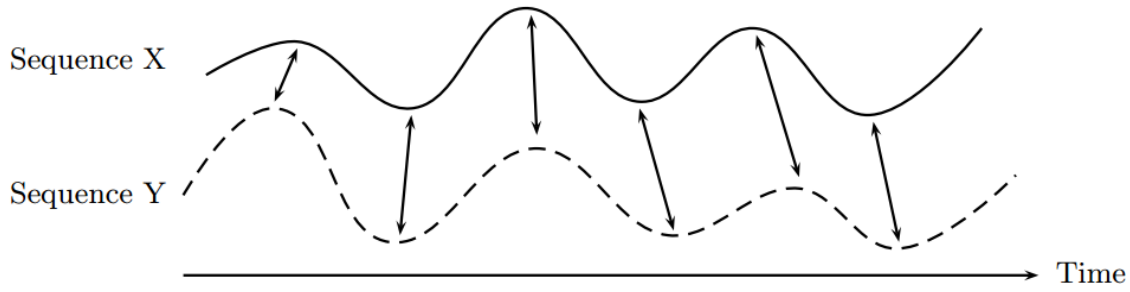

Figure 8 - Time alignment of two time-dependent sequences [29]

Intuitively, the sequences are warped in a nonlinear fashion to match each other. This technique is often used in automatic speech recognition, to handle different speaking speeds [29]. Other applications include speaker recognition [30] and on-line signature verification [31]. We can apply this technique to two dynamic gesture instances using the algorithm shown in Figure 9.

```
1  float DtwDistance(dg1, dg2) {
2    float[,] dtw = new float[dg1.Samples.Count, dg2.Samples.Count];
3    dtw.SetAllValues(PositiveInfinity); // Init each value to infinity.
4    dtw[0,0] = 0;
5
6    for(int i=0; i < dg1.Samples.Count; i++) {
7      for(int j=0; j < dg2.Samples.Count; j++) {
8        float cost = dg1.Samples[i].DistanceTo(dg2.Samples[j]);
9        cost += Min{ dtw[i-1,j], dtw[i,j-1], dtw[i-1,j-1] };
10       dtw[i,j] = cost;
11     }
12   }
13
14   // Now backtrack through and find the path (need to know its length).
15   Stack<int[,]> path = new Stack<int[,]>();
16   int x = dg1.Samples.Count - 1;
17   int y = dg2.Samples.Count - 1;
18   path.Push(new int[] {x, y} );
19   while(x > 0 || y > 0) {
20     if(x == 0) y--;
21     else if(y == 0) x--;
22     else {
23       float minNeighborCost = Min { dtw[x-1,y-1], dtw[x-1,y], dtw[x,y-1] };
24       if(dtw[x-1,y] == minNeighborCost) x--;
25       else if(dtw[x,y-1] == minNeighborCost) y--;
26       else { x--; y--; }
27     }
28     path.Push(new int[] {x, y});
29   }
30
31   return dtw[dg1.Samples.Count - 1, dg2.Samples.Count - 1] / (float)path.Count;
32 }
```

Figure 9 - Pseudocode for calculating the DTW distance between two dynamic gestures

This algorithm works for dynamic gestures of varying sample count, with no need to resize. In theory, it should result in smaller distances than the linear interpolation approach, as it looks for an optimal matching between the two sequences rather than directly mapping one sequence to the other. It is a dynamic programming algorithm, as it builds upon smaller DTW values that were computed earlier (see lines 6-12 of the pseudocode in Figure 9). The run time is *O(NM)*, where *N* and *M* are the sample sizes of the two gestures.

Graphical User Interface

The screen space is divided into five sections: (1) the menu bar, (2) the gesture library panel, (3) the recognition monitor / edit gesture panel, (4) the output window, and (5) the OpenGL window. These sections are labeled in Figure 10 - Labeled screenshot of the application in recognize mode.

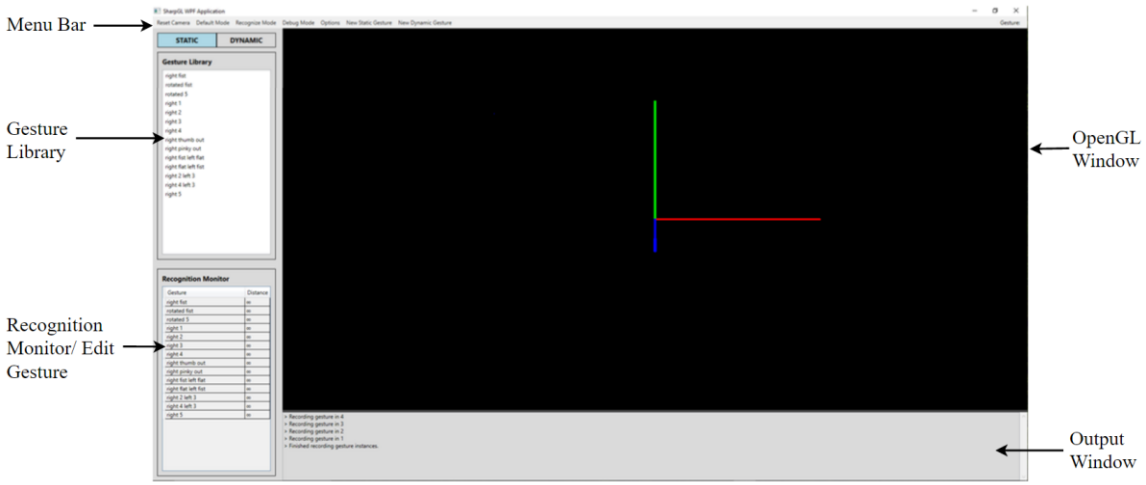

Figure 10 - Labeled screenshot of the application in recognize mode

Menu Bar

Table 5 shows each of the menu bar items and their actions.

Table 5 - Menu bar items and their actions

Menu Item	Action
Reset Camera	Resets the camera to its initial position.
Recognize Mode	Switches to recognize mode.
Options	Opens the “Options” modal window. Here the user can toggle boolean options, such as “Show Arms” and “Show Axes”, and set the colors of individual bones
New Static Gesture	Create a new static gesture.
New Dynamic Gesture	Create a new dynamic gesture.

Gesture Library

The gesture library displays a list of all saved gestures. The user can switch between static and dynamic gestures by clicking the appropriate tab (“Static” or “Dynamic”). Right clicking on a gesture opens a context menu allowing the user to edit or delete. Selecting “Edit” (or alternatively, double clicking the gesture name) displays an instance of the gesture in the OpenGL window and loads the “Edit Gesture” window for that gesture. Selecting “Delete” deletes the gesture and all of its instances from the database. Figure 11 shows a screenshot of the gesture library, with the context menu opened for the gesture “right pinky out”.

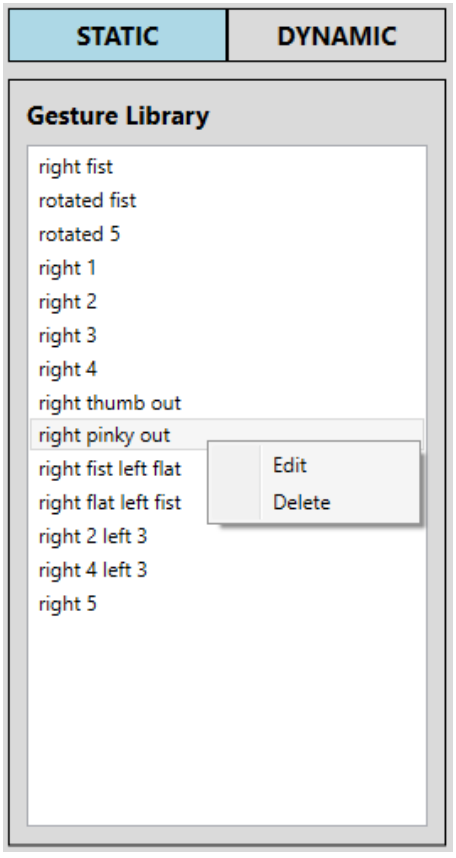

Figure 11 - Gesture Library

Recognition Monitor

The recognition monitor is only visible when in “Recognize” mode. It displays a two-column grid of gesture classes and the distance the most recent gesture instance is from that gesture class, ordered by ascending distance. A distance of infinity is displayed if the number of hands in the gesture class does not match the number of hands in the most recent instance. If no hands are in range of the Leap device the list does not update. Figure 12 shows snapshots of the recognition monitor in static (left) and dynamic (right) mode. Recognized gestures are highlighted in yellow.

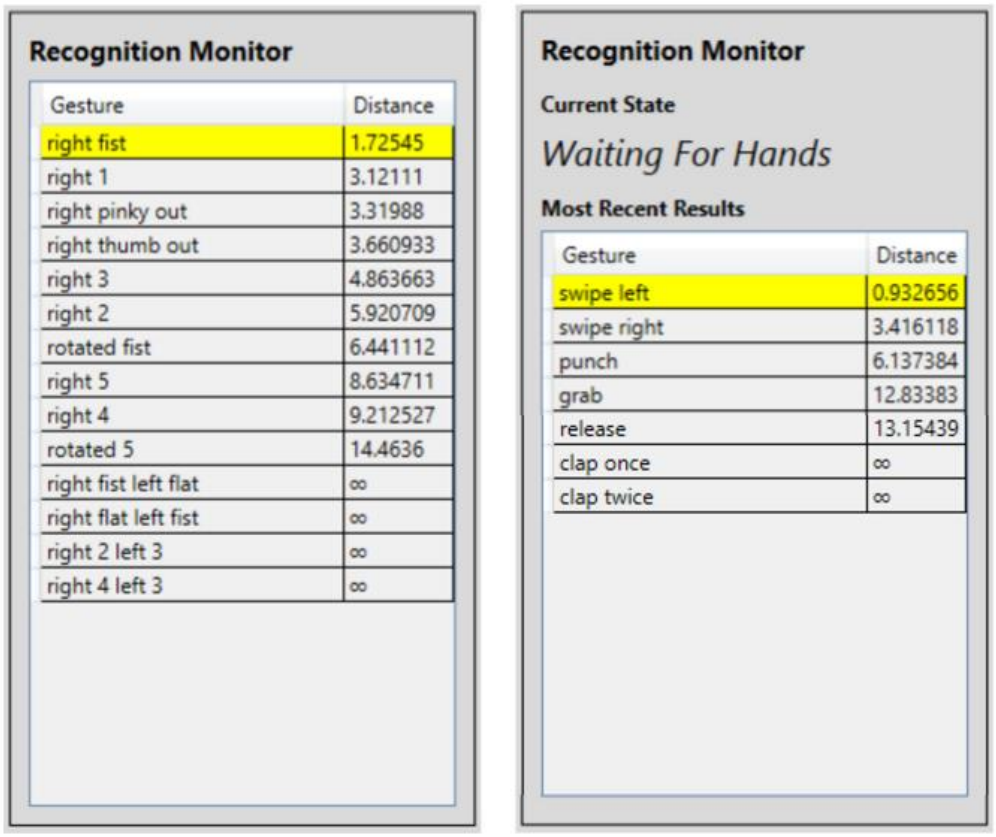

Figure 12 - Recognition monitor for static (left) and dynamic (right) gestures

In static mode, the recognition monitor updates continuously, comparing the live gesture instance to all existing static gesture classes. In dynamic mode, the recognition monitor works as a state machine, recording dynamic gestures performed by the user. The current state is displayed on the recognition monitor (see Figure 12). Figure 13 shows how this state machine works.

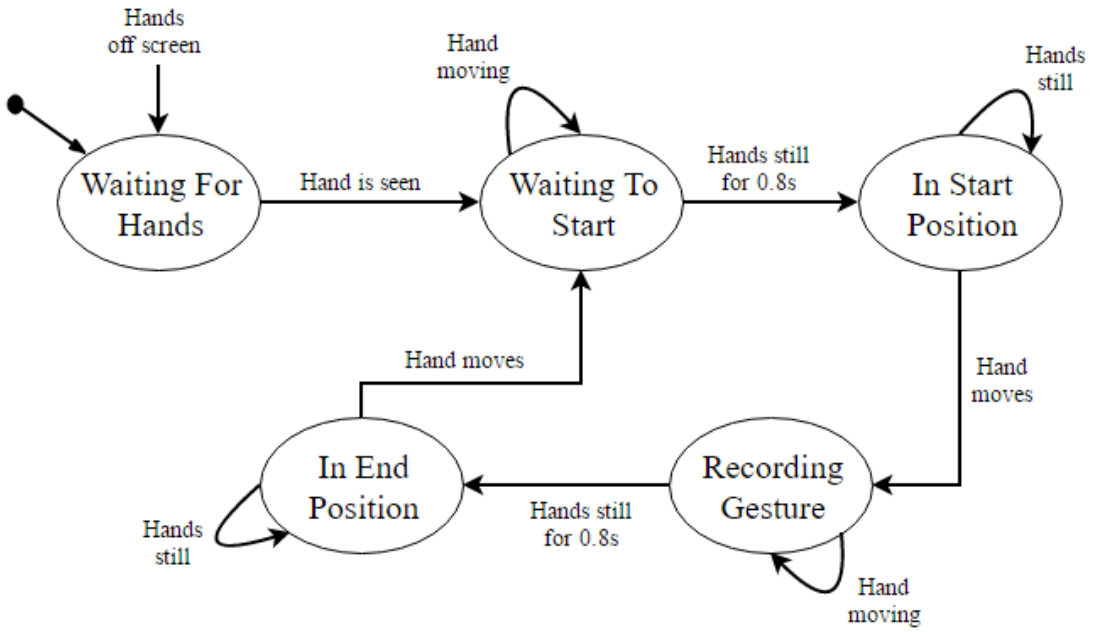

Figure 13 - Dynamic gesture recorder state machine

Upon transition to the state “In End Position”, a new dynamic gesture instance has just finished recording and is compared to all existing classes. The distance rankings are then updated in the recognition monitor.

Edit Gesture

The “Edit Gesture” window allows the user to rename a gesture, view or delete each of its instances, record new instances, or adjust feature weights. The gesture name is displayed in an editable textbox on top, allowing for easy renaming. Below that is a scrollable list of each of the instances of the gesture. Right clicking on one of the instances opens a context menu allowing the user to view or delete. Selecting “View” (or double clicking the instance name) displays that instance in the OpenGL window. Selecting “Delete” deletes the instance, which is useful for removing any outliers. Each feature’s weight can be adjusted by changing the value in the appropriate text box. Features have a weight of 1 by default. Clicking “Record Gesture” starts a session for recording new instances of that gesture. Details of the recording session are printed to the output window. All of the new instances show up in the instances list as the recording goes on. None of the changes (rename, delete instance(s), add new instance(s), adjust feature weights) are saved until the user confirms by clicking the “Save” button. Clicking the “Cancel” button closes the “Edit Gesture” window, discarding all changes. The application goes back into “Recognize” mode when the editing session is completed. Figure 14 shows a screenshot of the edit gesture window opened for the static gesture “right fist”.

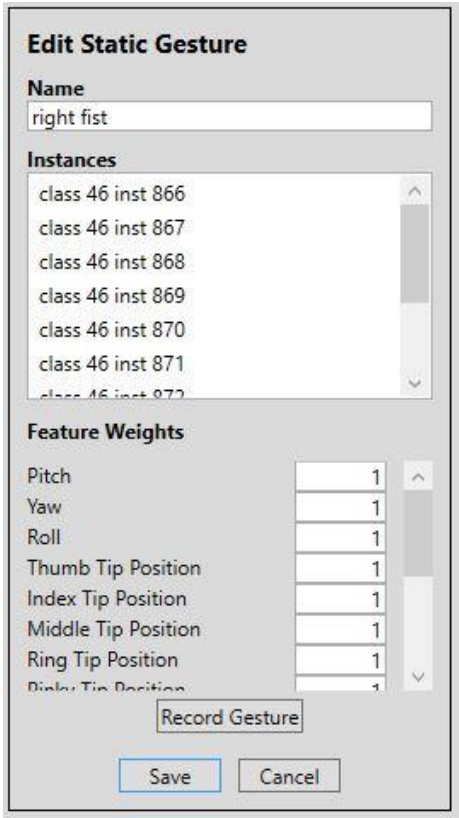

Figure 14 - Edit (static) gesture

Output Window

The output window is used for communicating various information to the user. This includes information about the recording session and other debug information.

OpenGL Window

The OpenGL window displays all 3D graphics, including the hands and the optional 3D axes. The 3D axes exist for the user's convenience. The intersection of the axes corresponds to the position of the Leap Motion controller. Figure 15 shows the 3D axes next to a left hand (red is +X, green is +Y, and blue is +Z).

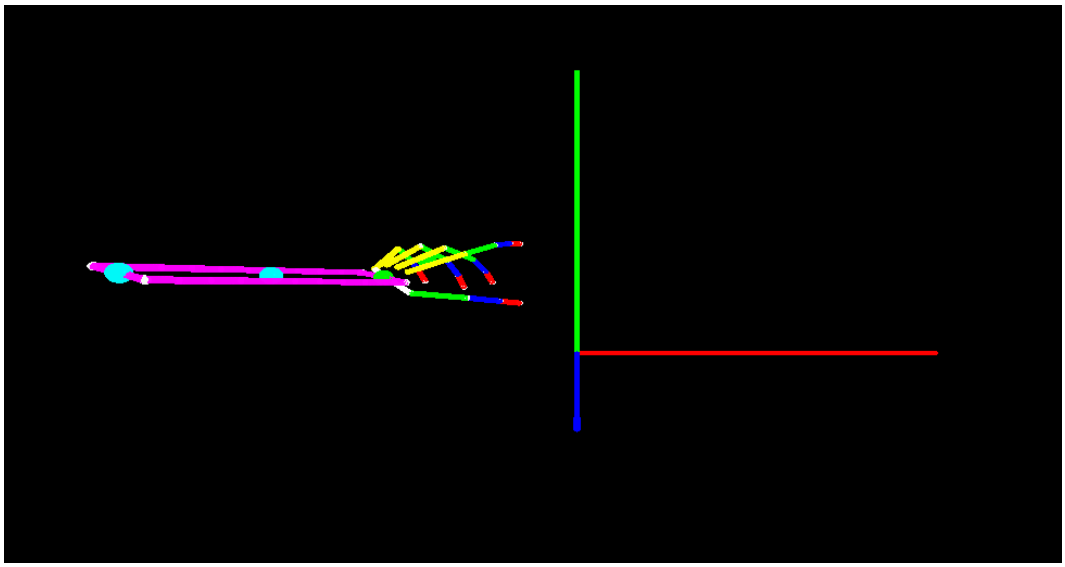

Figure 15 - Optional 3D axes

The user can rotate the camera in an arcball fashion by clicking the scroll wheel and moving the mouse around, or by using the arrow keys. The camera can be zoomed in and out using the mouse's scroll wheel. Figure 16 shows four different angles of an instance of the static gesture "flat right hand".

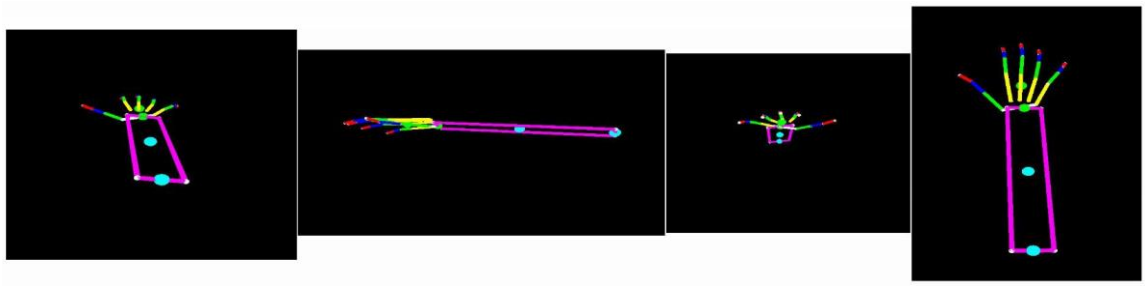

Figure 16 - Various angles of the static gesture "flat right hand"

Options Modal Window

A modal window pops up when the user clicks “Options” in the menu bar. There are two separate tabs the user can view. First is “General Options”, which includes the boolean options “Show Arms” and “Show Axes” (shown in Figure 17). Second is the “Bone Colors” tab, which allows the user to set the colors of individual bones (shown in Figure 18). The ColorPicker control was taken from the Extended WPF Toolkit [15].

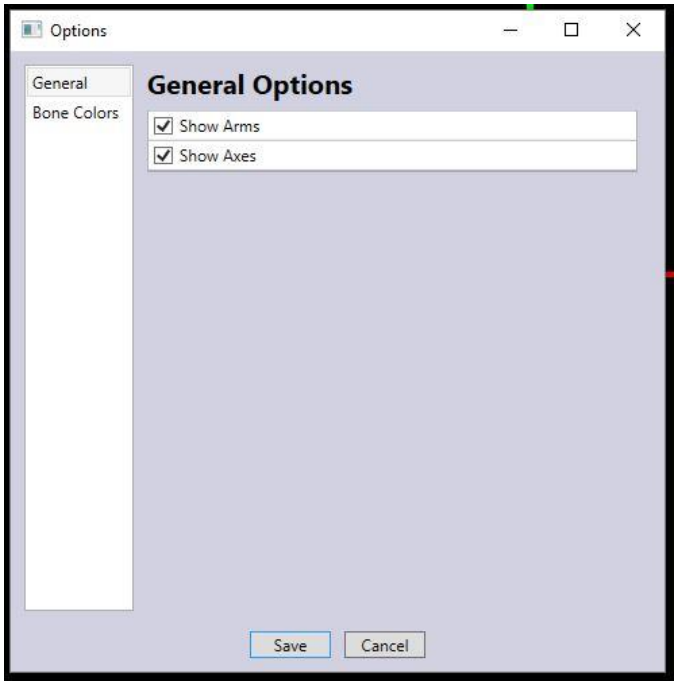

Figure 17 - General Options

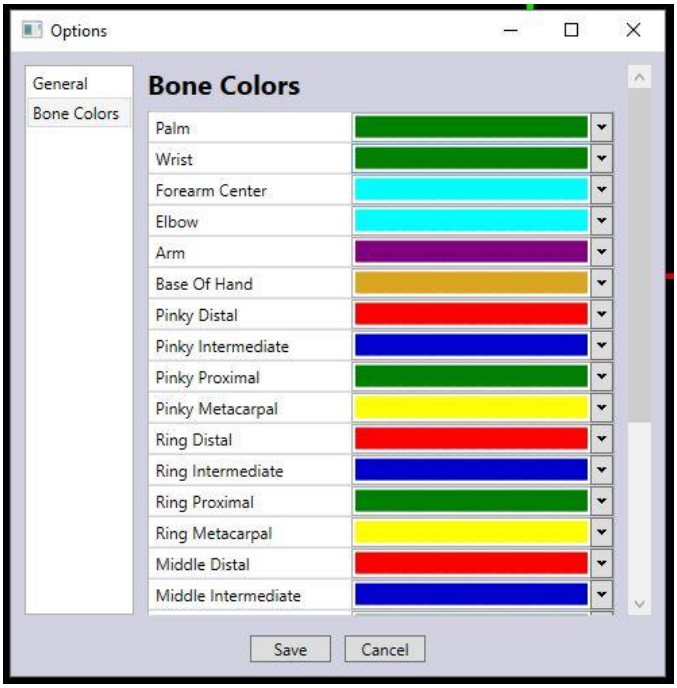

Figure 18 - Bone Colors

Experimental Results

To quantify the success of the system, I needed to generate some data. Two separate test participants were involved in this process. One for the training phase (building a library of gestures) and the other for the recognition phase. The training participant had some prior experience using the application (and the Leap Motion Controller in general), while the recognition participant did not. Figure 19 and Figure 20 show the static and dynamic gestures chosen for this process. Note: The camera angle is not consistent throughout all images; the angle with the best view was chosen for each gesture. The Leap Motion Controller was placed face up, on top of a desk for the entire testing process. In the training phase, participant 1 performed each gesture 5-10 times. Each of these training sets was manually inspected to remove or replace any bad samples, and to ensure an accurate distribution of acceptable variations for each gesture. In the recognition phase, participant 2 performed each gesture three times. The average distance to the target gesture was recorded, along with any other gestures that were also recognized.

Static Gestures

Figure 19 shows the static gestures selected for the testing process.

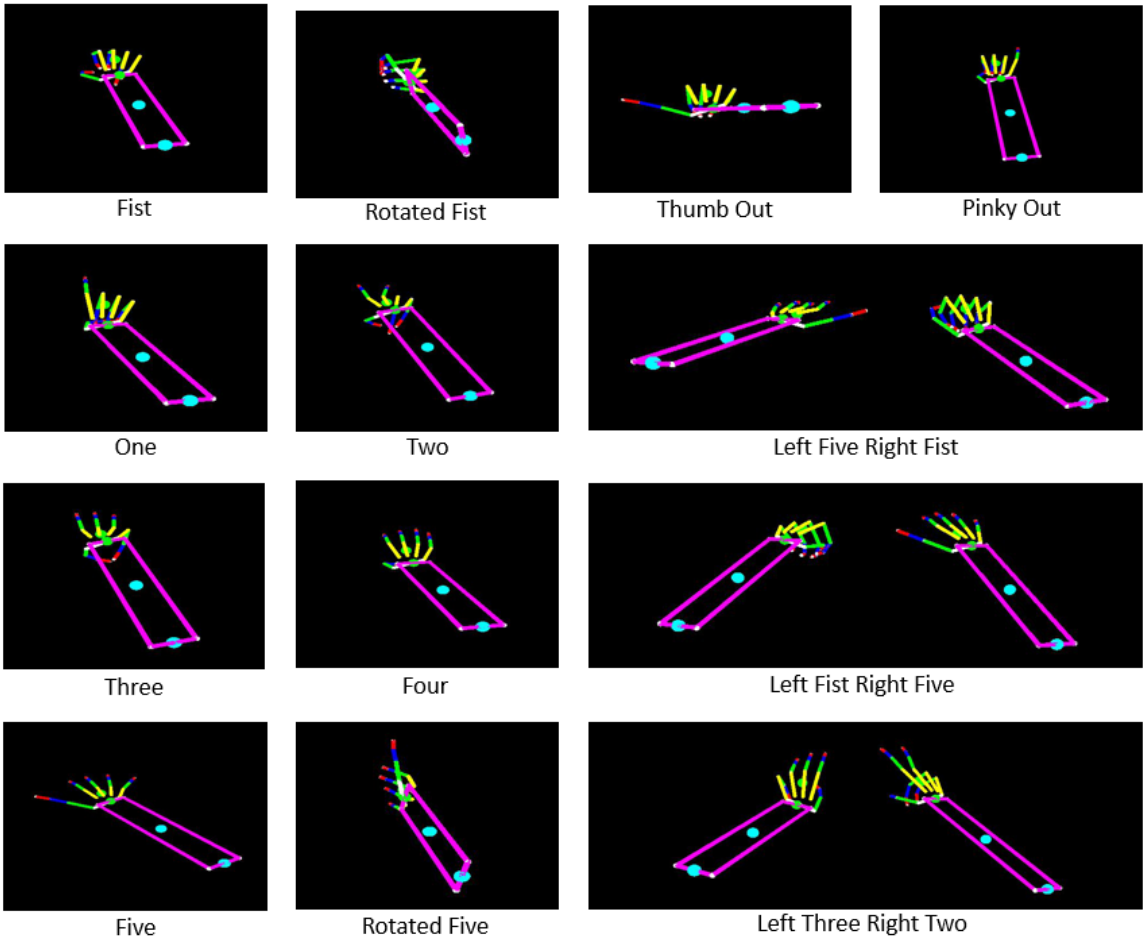

Figure 19 - Static gestures used in testing process

Table 6 – Experimental results for static gesture recognition lists the recognition testing results for each of the static gestures shown in Figure 19.

Table 6 – Experimental results for static gesture recognition

Static Gesture	Distance	Other Gestures Recognized
Fist	1.87	Thumb Out (2.65), One (2.84)
Thumb Out	1.9	
Pinky Out	1.83	
One	1.83	Fist (2.7)
Two	1.9	Three (2.22)
Three	1.47	
Four	1.9	Three (2.25), Five (2.5)
Five	1.2	Four (2.7)
Rotated Fist	2.27	
Rotated Five	2.27	
Right Fist Left Flat	2.32	
Right Flat Left Fist	2.43	
Right Two Left Three	1.73	

The “Distance” column contains the average distance the performed gesture was to the target gesture class. For the “Fist” gesture, for example, the three performances of the gesture had distances of 1.5, 1.6, and 2.5 (an average of 1.87) from the trained “Fist” class. As mentioned in the “Static Gesture Recognition” section, a distance of 3.0 or less is considered a match. Every gesture in Table 6 was recognized. The two gestures “Rotated Fist” and “Rotated Five” had larger distance compared to “Fist” and “Five” likely because it is harder to for the Leap Motion Controller to track the hands at that

angle, resulting in inconsistent data. Similarly, the gesture “Five” had the smallest distance likely because all fingers are easily visible to the Leap Motion Controller, making the hand easier to track. In a few cases, other gestures were also under the threshold distance of 3.0. This makes sense, as some of the gestures are very similar to each other. None of these other gestures had smaller distance than the target gesture. Although they are considered recognized, in every case the target gesture is an even closer match. All things considered, the static recognition system exceeded expectations.

Dynamic Gestures

Figure 20 shows the dynamic gestures that were chosen. Each row shows the start, middle, and end snapshots of the gesture (more samples exist in between).

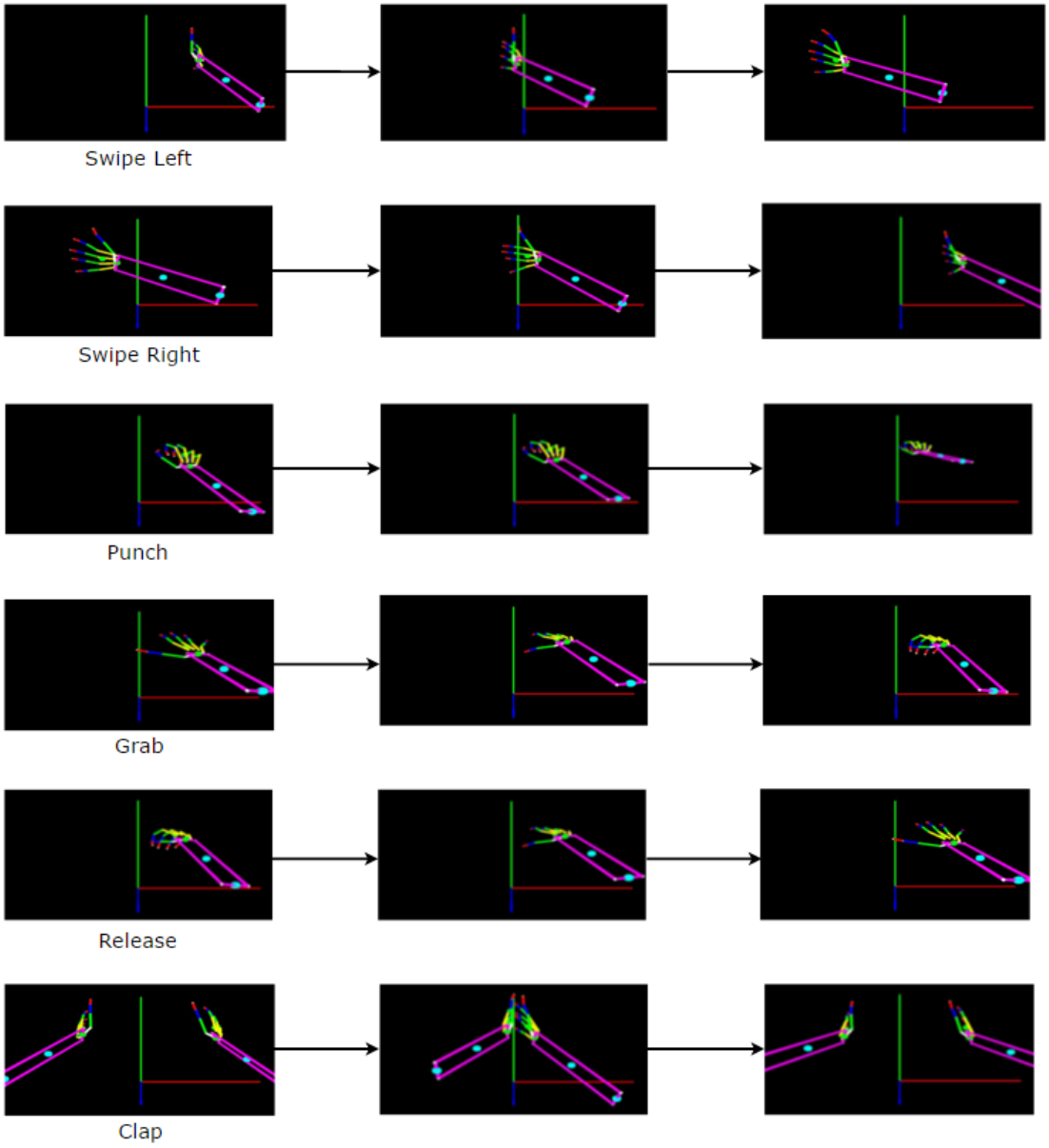

Figure 20 - Dynamic gestures used in testing process

Table 7 shows the results for the dynamic gestures of Figure 20. Both the linear interpolation (lerp) and dynamic time warping (DTW) distance methods were tested (see "Dynamic Gesture Recognition" for more information on these two approaches).

Table 7 - Experimental results for dynamic gesture recognition

Dynamic Gesture	Lerped Distance	Other Gestures Recognized Using Lerp Approach	DTW Distance	Other Gestures Recognized Using DTW Approach
Swipe Left	1.85		2.84	
Swipe Right	1.2		1.12	
Punch	2.28		1.53	
Grab	2.19		2.4	Punch (2.78)
Release	2.29		2.77	Punch (2.81)
Clap	2.72		2.81	

Just as in the static gesture tests, the "Distance" columns contain the average distance the performed gesture was to the target class. A distance of 3.0 or less is considered a match. The lerp approach worked quite well: all target gestures were recognized, and no other gestures were mistakenly recognized. The DTW approach was also successful, but it did result in some false positives. "Grab" and "Release" were also recognized as "Punch". Both gestures start or end in the same posture as "Punch" (see Figure 20), so this is not surprising, especially with dynamic time warping looking for an optimal mapping between the gestures.

During general use of the application, it was noticed that the lerp implementation is more sensitive to speed and overall gesture duration. Slow and fast versions of a gesture are not recognized as the same. The DTW implementation, on the other hand, is not as sensitive to time and speed. It is able to recognize the same gesture at varying speeds. The DTW approach also resulted in more false positives than the lerp approach. Overall, the lerp approach tended to be more accurate than the DTW approach. However, the DTW implementation could likely be improved to work just as well, if not better.

Implementation

As mentioned earlier, the application was built using WPF, following the MVVM design pattern. Table 8 and Table 9 describe the C# classes that fall under the "Model" and "ViewModel" categories of the MVVM pattern. Each row includes the name, lines of code (LOC), and a brief description of the class. The LOC values were approximated using Visual Studio 2013's Code Metrics tool [35].

Table 8 - Classes that fall under the "Model" category of the MVVM pattern

Class Name	LOC	Description
`SGClass`	255	Static gesture class. Responsible for computing the mean and standard deviation of each feature of a gesture, given a list of static gesture instances. Uses this data to calculate distance from a static gesture instance.
`SGInstance`	206	Static gesture instance. Made up of two `SGInstanceSingleHand`'s.
`SGInstanceSingleHand`	156	Static gesture instance single hand. Contains all data and information relevant to a single hand. Includes data used in `SGInstance`'s feature vector, and also additional data for drawing the hands.

SGRecorder	48	Static gesture recorder. Contains methods and properties for recording static gesture instances.
DGClass	140	Dynamic gesture class. Responsible for building a list of DGClassSamples given a list of DGInstances. Uses this data to calculate distance from a DGInstance.
DGClassSample	46	Dynamic gesture class sample. An extension SGClass. Also includes velocity data. Given a list of DGInstanceSamples, calculates the mean and standard deviations of each feature. Uses this data to calculate the distance from a DGInstanceSample.
DGInstance	74	Dynamic gesture instance. Consists of a list of DGInstanceSamples and methods for computing the distance to other DGInstances.
DGInstanceSample	24	Dynamic gesture instance sample. An extension of SGInstanceClass, also including velocity information.
DGRecorder	74	Dynamic gesture recorder. Contains methods and properties for recording dynamic gestures.
SharpGLHelper	111	A set of helper methods for drawing the hands and other entities in the OpenGL window, using the SharpGL library.
SQLiteProvider	172	Provides a layer of abstraction between the application logic and the SQLite database. Contains methods for accessing and manipulating the stored data.
Camera	65	Responsible for handling movement of the OpenGL window's camera, including methods for zooming and revolving around the origin of the scene.
All other classes…	285	All of the other smaller, Model classes that were implemented but not worthy of individual mention.
TOTAL	1,656	All Model classes combined.

Table 9 - Classes that fall under the "ViewModel" category of the MVVM pattern

Class Name	LOC	Description
MainViewModel	268	Contains general application logic and properties.
EditStaticGestureViewModel	86	Contains logic and properties related to the editing of static gestures.

`EditDynamicGestureViewModel`	77	Contains logic and properties related to the editing of dynamic gestures.
`RecognitionMonitorViewModel`	52	Contains logic and properties related to the recognition of static and dynamic gestures.
`OptionsViewModel`	29	Contains logic and properties relevant to the options window.
TOTAL	512	All "ViewModel" classes combined.

Most of the UI ("view") code was written in XAML, though there was some additional code-behind written in C#. Table 10 describes the XAML files that make up the "View" portion of the project. The "Tags" column displays the number of XAML tags in the file, giving a measurement of its size (similar to lines of code).

Table 10 - XAML files that define the user interface

File	Tags	Description
MainWindow.xaml	56	Defines the overall application layout. As the highest-level window element, it instantiates all custom controls (listed in this table) as children. Bound to `MainViewModel`.
EditStaticGesture.xaml	40	Defines the UI for editing static gestures. Bound to `EditStaticGestureViewModel`.
EditDynamicGesture.xaml	42	Defines the UI for editing dynamic gestures. Bound to `EditDynamicGestureViewModel`.
GestureLibrary.xaml	28	Defines the UI for interacting with the gesture library (all saved static and dynamic gestures). Bound to `MainViewModel`.
RecognitionMonitor.xaml	38	Defines the UI for static and dynamic gesture recognition. Bound to `RecognitionMonitorViewModel`.
Options.xaml	42	Defines the UI for the options window. Bound to `OptionsViewModel`
TOTAL	246	Total number of tags in all XAML files.

Table 11 lists each of the code-behind files in the “View”. A total of 173 lines of code were written for these files, a number that could have been reduced if the MVVM pattern was followed more closely.

Table 11 - "View" code-behind files

File	LOC	Description
MainWindow.xaml.cs	25	Code-behind for MainWindow.xaml.
EditStaticGesture.xaml.cs	39	Code-behind for EditStaticGesture.xaml
EditDynamicGesture.xaml.cs	36	Code-behind for EditDynamicGesture.xaml
GestureLibrary.xaml.cs	36	Code-behind for GestureLibrary.xaml
RecognitionMonitor.xaml.cs	11	Code-behind for RecognitionMonitor.xaml
Options.xaml.cs	26	Code-behind for Options.xaml
TOTAL	173	All “xaml.cs” files combined.

CONCLUSION

Summary

This project successfully demonstrates the effectiveness of a statistical pattern recognition system for identifying static and dynamic hand gestures. Given minimal training data, the system was able to consistently recognize each of the test gestures shown in Figure 19 - Static gestures used in testing process and Figure 20. Separate test participants were used for the training and recognition phases.

The simplicity of the Leap Motion Controller and its API make the recognition process straightforward compared to many of the existing vision or glove based techniques. Though the Leap Motion Controller technically does use a vision-based approach, this step is abstracted away from the user: tracking data is provided in a clean and accessible format. The device is not perfect, as it cannot always correctly track fingers that are occluded by other objects, and its range is limited, among other things; but the technology is continuing to improve.

Future Improvements and Ideas

Though I do consider the project a success, there are many things that could be improved. First off are some appearance-based improvements. The GUI is perfectly functional and doesn't look bad, but the style could be improved. The 3D rendering of the hands could be improved to look more human-like. Right now they are just made up of cylinders and spheres.

Another area of improvement is in dynamic gesture recognition. As of now, the user needs to hold still for a small delay, then perform the gesture, then hold still again. This is done to identify the start and the end of the gesture. This isn't practical for many real applications. Ideally, the user would continuously move their hands and the software would be able to identify the start and end of a gesture on its own.

The number of "false positives" (other gestures mistakenly recognized) could be reduced by making similar gestures more distinguishable from one another. Currently this can be done by editing the gesture and manually setting the weights (see Figure 14), but it would be nice to have this process automated. Static gestures "Four" and "Five" in Figure 19, for example, were both recognized mistakenly when the other was performed (see Table 6 – Experimental results for static gesture recognition). As a fix, I could write a method that first goes through the gesture library looking for pairs of gestures that are very similar to each other (small distance between them). It then compares the feature vectors of both gestures, increasing the weight of any features that are significantly different from each other. This increases the distance between the two gestures. The method is not finished at this point; it must then must compare these two modified gestures to each of the others again, to see if any more adjustments need to be made. As an example, consider gestures "Four" and "Five" in Figure 19. The major difference between them is the pinky tip position feature. Its weight should be increased for both gestures, resulting in an increased distance between the two. This method should be called each time new gesture instances are saved.

A single Leap Motion Controller has its limits, as there is no way for the device to track fingers that are occluded by other regions of the hand. As more of a separate project, I would like to use multiple Leap Motion Controllers in unison, to capture the hands from different angles and reduce the chance of hand regions being occluded. One device could be placed face up on the desk and another mounted to the user's forehead. The devices would need to be synced up and their data combined to produce more accurate tracking data.

Finally, I would like to make it easy for people to integrate this gesture recognition system into their own projects. Users would download this application as a standalone tool for building a library of their own custom gestures. Then, in their own projects, they could reference a DLL that provides classes and methods for recognizing gestures in the format produced by this application.

REFERENCES

1. Bellugi, Ursula. The signs of language. Harvard University Press, 1979.

2. Sandler, Wendy, and Diane Lillo-Martin. Sign language and linguistic universals. Cambridge University Press, 2006.

3. Pavlovic, Vladimir I., Rajeev Sharma, and Thomas S. Huang. "Visual interpretation of hand gestures for human-computer interaction: A review." Pattern Analysis and Machine Intelligence, IEEE Transactions on 19.7 (1997): 677-695.

4. https://www.leapmotion.com Accessed on February 28, 2016.

5. Quek, Francis KH. "Toward a vision-based hand gesture interface." Virtual Reality Software and Technology Conference. Vol. 94. 1994.

6. Mulder, Axel. "Hand Gestures for HCI: research on human movement behaviour reviewed in the context of hand centred input." Simon Fraser University, Canada (1996).

7. Premaratne, Prashan. "Historical Development of Hand Gesture Recognition." Human Computer Interaction Using Hand Gestures. Springer Singapore, 2014. 5-29.

8. Dipietro, Laura, Angelo M. Sabatini, and Paolo Dario. "A survey of glove-based systems and their applications." Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 38.4 (2008): 461-482.

9. https://msdn.microsoft.com/en-us/library/hh438998.aspx Accessed on February 28, 2016.

10. http://blog.leapmotion.com/getting-started-leap-motion-sdk Accessed on February 28, 2016.

11. https://msdn.microsoft.com/en-us/library/mt149842.aspx Accessed on March 1, 2016.

12. https://sharpgl.codeplex.com Accessed on March 1, 2016.

13. http://www.codeproject.com/Articles/3144/SharpGL-a-C-OpenGL-class-library Accessed on March 1, 2016.

14. https://www.sqlite.org Accessed on March 1, 2016.

15. https://www.leapmotion.com/product/desktop Accessed on March 1, 2016.

16. http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work Accessed on March 1, 2016.

17. Jain, Anil K., Robert PW Duin, and Jianchang Mao. "Statistical pattern recognition: A review." Pattern Analysis and Machine Intelligence, IEEE Transactions on 22.1 (2000): 4-37.

18. https://developer.leapmotion.com/documentation/csharp/api/Leap_Classes.html Accessed on March 1, 2016.

19. http://wpftoolkit.codeplex.com/ Accessed on March 1, 2016.

20. Marin, Giulio, Fabio Dominio, and Pietro Zanuttigh. "Hand gesture recognition with leap motion and kinect devices." Image Processing (ICIP), 2014 IEEE International Conference on. IEEE, 2014.

21. Potter, Leigh Ellen, Jake Araullo, and Lewis Carter. "The leap motion controller: a view on sign language." Proceedings of the 25th Australian computer-human interaction conference: augmentation, application, innovation, collaboration. ACM, 2013.

22. T. Takahashi and F. Kishino. Hand gesture coding based on experiments using a hand gesture interface device. pages 67– 73, 2003.

23. Parvini, Farid, et al. "An approach to glove-based gesture recognition." Human-Computer Interaction. Novel Interaction Methods and Techniques. Springer Berlin Heidelberg, 2009. 236-245.

24. Newby, Gregory B. "Gesture recognition using statistical similarity." Proceedings of virtual reality and persons with disabilities (1993).

25. Hong, Pengyu, Matthew Turk, and Thomas S. Huang. "Constructing finite state machines for fast gesture recognition." Pattern Recognition, 2000. Proceedings. 15th International Conference on. Vol. 3. IEEE, 2000.

26. https://system.data.sqlite.org Accessed on April 5, 2016.

27. https://github.com/dwmkerr/sharpgl Accessed on April 5, 2016.

28. http://statweb.stanford.edu/~naras/jsm/NormalDensity/NormalDensity.html Accessed on April 7, 2016.

29. Müller, Meinard. "Dynamic time warping." Information retrieval for music and motion (2007): 69-84.

30. Yu, Kin, John Mason, and John Oglesby. "Speaker recognition using hidden Markov models, dynamic time warping and vector quantisation." *Vision, Image and Signal Processing, IEE Proceedings*-. Vol. 142. No. 5. IET, 1995.

31. Martens, Ronny, and Luc Claesen. "On-line signature verification by dynamic time-warping." *Pattern Recognition, 1996. Proceedings of the 13th International Conference on*. Vol. 3. IEEE, 1996.

32. https://en.wikipedia.org/wiki/Dynamic_time_warping Accessed on April 10, 2016.

33. https://msdn.microsoft.com/en-us/library/ms754130(v=vs.100).aspx Accessed on April 11, 2016.

34. https://msdn.microsoft.com/en-us/library/hh848246.aspx Accessed on April 13, 2016.

35. https://msdn.microsoft.com/en-us/library/bb385914.aspx Accessed on April 13, 2016.

36. https://developer.leapmotion.com/documentation/csharp/devguide/Leap_Overview.html Accessed on April 13, 2016.